

[nominal delivery draft]

On Measurement, Daniel E. Geer, Jr., Sc.D.
Software Assurance Forum, Mclean, Virginia, 19 Sept 12

ABSTRACT:

The most important thing the Software Assurance Community can do is to ensure that there is no silent failure. This means instrumentation, it means well designed surveillance regimes, it means an attention to the kind of metrics that come out of an airplane's flight data recorder, it means keeping things simple enough that, well, there are fewer surprises, and it may mean changing how you think about how you make tradeoffs. Repeating Kernighan, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. Because security is not composable (and may never be), be very careful where the code you reuse comes from.

Good afternoon. As always, I am both appreciative of, and frightened by, the opportunity to stand before my colleagues and give advice. There was a time -- a time not so very long ago -- when the number of people who had a care about software security might be countable. This is no longer the case; there are many people working on the issue. We are, for almost all values of "we," spending a great deal more money and time on software security than once we were. Yet while we are winning battles, we are not winning the war.

Perhaps we never will. Perhaps the rate of change that we have come to know and love in the digital sphere is itself a guarantee of an unrestrained climb in the workfactor that software security requires.

Last February, in a different setting, I made the case [1] for choosing between two roads while we were still at the intersection of them -- the one was to damp down the growth of complexity such that human-scale responses to untoward changes in the computing environment remained the best kind of responses to make. The other was to conclude that the time had come to pick automation of cybersecurity as the way forward. The thesis behind this dichotomy is obvious; there comes a point in the growth of scale and complexity where no longer is it possible for informed decision making of the human sort.

Note that I said "informed" decision making. Decision making is always possible, but informed decision making is another thing altogether. We have actually had numerous straightforward proofs of this thesis, one of which may have been the first even if it was not recognized as such at the time. When the Three Mile Island nuclear reactor failed in 1979, the lesson was that in a fully instrumented control room, human decision making is paralyzed if enough of the alarms go off at once. A few flashing lights, and everyone sprints to their station and starts dialing up remediations to quench the flashing lights, but if enough lights are flashing, everyone freezes and there is no remediation.

...

For the purposes of this discussion, I rather favor Peter Bernstein's definition of risk as "more things can happen than will." The bland neutrality of that definition helps me think about risk and remediation of risk as one where preparation matters more than courage. Bravery has its place, and let us never forget the honor deserved by those who have placed the welfare of others ahead of their own, but cybersecurity has much more room for preparation than it does for bravery.

In "Against the Gods," [2] Bernstein convincingly made the point that the modern world itself dates from the time when we humans came to see risk as something that could be described, something that could be measured, something that could be anticipated -- in short, that the analysis of risk was not only feasible but empowering. Bernstein's bibliography, or the history of insurance, teach us that understanding risk frees one from a fatalism that can only rob us of initiative.

As with all people my age in cybersecurity, I was trained for something else. In my case, it was biostatistics which is to say decision making under uncertainty with mortal consequences. There are many other preparations that would do as well; civil engineers study why bridges fall down, attorneys study how to write policy that can be put into actual practice, and military field hospitals are from whence the concept of triage comes.

In the pantheon of statistical science, there is, on the one hand, the legacy of Ronald Fisher and the frequentist school and, on the other hand, the legacy of Rev. Thomas Bayes whose name is on the bayesian school. The frequentists argue that only things which repeat can be estimated well enough to inform planning for the future -- that one time events do not and cannot contribute. The bayesians argue that the true probability of anything is not the point but, rather, each bit of data can and should change our beliefs of what the future might bring. The proportion of your time you spend as a frequentist or as a bayesian probably depends on how uncertain are the parameters of the decisions you must make. With no data at all, you have only your beliefs to go on. With a deluge of data about a physical law, the frequency of an event is definitive. It is in between that is more challenging. It is in between that we find ourselves.[3]

As I said in the Abstract, the most important thing the Software Assurance Community can do is to ensure that there is no silent failure. This means instrumentation, it means well designed surveillance regimes, it means an attention to the kind of metrics that come out of an airplane's black box, it means keeping things simple enough that there are ever fewer surprises, and it may mean changing how you think about making tradeoffs.

Software as a service illustrates the tradeoff challenge. When you believe that your code won't actually be seen by its users because they are only buying it as a service, your tendency will be to compete not on ease of installation, update, field supportability or integrability, but rather on performance and the latency of

Left to themselves, creative engineers will deliver the most complicated system they think they can debug. [4]

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it? [5]

That may be good in the end, but we might have already seen version 1.0 of the strategy of turning decision making over to machines -- it's called High Frequency Trading and, in a proof by demonstration, those who implemented it did not know what the complexity and the speed would do to their algorithms. Nevertheless, the High Frequency Trading machines now dominate the trading arena, and bring us so-called Flash Crashes. Two stockbrokers who are proposing the simple rule that bids be honored for 50 milliseconds have been called cavemen and sore losers.[6]

• • •

MEW-6AEaEdEcAAlpcNi ~E^ eei e-aAElmcEiO of Lt ECaEc-dOEAc ~KQMNVKri

better decisions.

Does this idea of measurement for decision support apply to software assurance? I would not be here if I didn't think so, nor would you. What sorts of decisions need support in building better software? Are there measurements that, if made, might support better decisions? I think there are.

Good guys and bad guys alike have come a long way toward building systems that can assess code quality. Good guys do it one way and bad guys do it another, but both good guys and bad guys have the same purpose -- deciding how much effort to spend to reach their real goal. (This is probably where I should acknowledge that to my mind all security products are dual use.) Assessing code quality is a central theme of this event, and tools for measuring code quality are increasingly abundant. Are they perfect? Of course not. Do I care? Not much.

Why do I not care if the tools are really, really good or not? Because to a first approximation, software builders need a relative measure, not an absolute one, for the decisions they need to make. Whatever it is that you are measuring, when your measurement instrument is not very good that does not mean that it is useless. If that measurement instrument has reasonably constant errors, then the trend lines it produces are still relevant, just not the absolute values. At least at first, I probably don't care whether my programming staff is perfect but I may well care about two relative measures: "Are they getting better month over month?" and "How do they compare to their peers?" Imperfect, even noisy, measures can help with those two questions (and others like them). Don't let the best be the enemy of the good.

As a side note, if you are the designer of some measurement method and you discover, whether through experience or brainstorm, a better measurement method, then please don't just replace the old method with the new one. Run them both side by side for a while. In that way, you will gain some ability to calibrate the new method against the old and, if all goes well, you will be able to avoid throwing away the trendlines that you have accumulated. If you just swap the new for the old, your record keeping starts again from scratch. That is not the way to do decision support.

...

Turning back toward complexity, when we measure software quality and/or security, [8] often as not our unit of observation is the software module, however defined. What we deploy in the field, however, is not a module but many modules linked together in various ways depending on language, base platform, and so forth. As an hypothesis, is the dominant fraction of exploitable security flaws at module interfaces? If that hypothesis is correct, then increasing numbers of modules creates risk. To the point of this meeting, I'd be very interested in some one of you doing a thoughtful, real-numbers analysis of the impact of cloud computing's cheapness on code bloat. Chris Wysopal, who speaks later, has in fact observed that the size of applications tends to rise after they are moved into the cloud precisely because space becomes too cheap to meter -- developers

link against any library that contains even one call they care about and, in any case, it's faster to just include everything. Bloat like that doesn't matter to anyone except, perhaps, us security people, but measuring it seems a worthy bit of decision support to me.

Another, shall we say, "question" is where is code executed? Mitja Kolsek [9] suggests that the way to think about the execution space on the web today is that the client has become the server's server. His comment does rather make a crucial point about software security, namely that the execution space is not one that the software writer has seen or will ever see. It is thus another irony, at least for measurement people, that the most common use of Javascript is the "measurement" that Google Analytics does with its ga.js script. Realistically speaking, if the execution space of your counterparty is where your code will run, then what began as "You're OK, I'm OK, but the network is dangerous" has become "I hope I'm OK, I have to assume that you are hosed, and the network may make this worse." Preparing for this is not so much a measurement problem as a detection problem -- a detection problem of paramount importance if you accept the notion from the abstract, that the most important thing the Software Assurance Community can do is to ensure that there is no silent failure.

...

The most telling legacy of Dennis Ritchie was that C had data structures, data structures that operated at a level that was just barely high enough. I've come to view parsimony of expressiveness as a talisman against silent failure. Let me quote Don Davis, [10] whose code is all but surely running on every computer in this room:

The network-security industry has produced lots of examples of over-rich expressiveness: RACF, firewall rules, and .htaccess are my favorite examples. I argue that in computer security applications, a language or UI should present a little less expressiveness than expert administrators will find necessary, so as not to help normal administrators to confuse themselves.

The problem is that every security rule-set has to be long-lived and to change steadily. If the rule-set's syntax allows for subtlety, then each rule-set's size and complexity tends only to grow, never to shrink. This is because each security administrator will tend to avoid analyzing whatever subtleties have accumulated, and will instead blindly add special-case allowances and constraints, so as to avoid breaking whatever came before. The typical result is an unwieldy rule-set that no human can understand, with unpredictable security holes.

Here, as remedy, are two rules of thumb: for security, avoid designing order-dependent syntax, and avoid recursive features, like groups of groups. Such features seem useful and innocuous, but when administrators use them heavily, complexity mounts destructively.

Don wrote that eight years ago. His distinction between programming language and what an administrator uses may now be a distinction

without a difference, but that does not disable his point; it strengthens it. PERL and Ruby and Java have too many ways to express the same thing, to do the same thing, and they brag about how "there is always another way." Each of the three are Turing complete, as is HTML5. Greater expressiveness seems to be the way things are going. That expressiveness means that it is not possible to ascertain whether a bit of software is or is not secure, that is to say that the question of its security is formally UNDECIDABLE. As mentioned above, that moves us from direct measurement of the security of software (it being impossible) into indirect measurement, just like finding a new planet.

But is indirect measurement of software security a mediocre strategy however much it might be the best we can do with the languages we are busy proliferating? That depends on what decisions your measurement is intended to support. If your decision is whether or not to deploy a piece of code, then your decision might will be supported by competent analysis of its security and quality. I do recommend the measurement of code quality by well-thought tools and there is a maturing group of suppliers of such toolsets.

Personally, I take a view centered on data security insofar as data is where the value is and data is, almost always, what attackers want in the end -- software is, dare I say, merely the vehicle for getting data. Repeating that the highest goal for the security professional is that there be no silent failure, then the actual decision to be made is whether data that wants to leave will be permitted to do so.

Here an old idea has become new. Jim Anderson published a paper[11] in 1972 that, by 1983, became the core of the Orange Book.[12] Anderson had invented the Reference Monitor. A Reference Monitor is a separate process that watches the first to make sure that it makes no data handling errors. This is an instructive idea, and in many ways is echoed in the many products involved in data and intrusion protection, namely that of a buttress to doing the best you can at designing applications in the first place. It is also a kind of measurement in the sense that it takes an observation from a privileged position and, perhaps in conjunction with other observations, supports a data handling decision. It is an example of "no silent failure" at work.

We have nowhere to go but up with respect to a rule of "no silent failure." The Verizon Data Breach Investigations Report shows that data loss is overwhelmingly silent. Part of that silence is digital physics -- if I steal your data, then you still have them, unlike when I steal your underpants -- but the majority of that silence is that there is no programmatic indicator of the data's cloning; it is like a (UNIX) `_fork_` operation, fast and cheap. [As an historical aside, the late Dennis Ritchie wrote that the "PDP-7's fork call required precisely 27 lines of assembly code."]

Together with colleague Mukul Pareek, we run the Index of Cyber Security, [13] a monthly measure of how people such as yourselves see the state of cybersecurity. We've been doing it for a year and a half now, and in the process measuring the sentiment of those with operational responsibility for their firms. In addition to a

fixed set of questions asked each month (just like our models, the Consumer Confidence Index and the Purchasing Managers' Index), we ask one additional question each month. The question for August was "Have you and/or your colleagues discovered an attack at another entity?" for which 55% said "Yes and confirmed" and another 10% said "Yes but unconfirmed." This is obviously an indirect measure, but it supports decisions about proper investment levels in software security based on the valuable data behind it. It also underscores that silent failure is happening, and matches up well with the Verizon DBIR.

...

The rapid rate of change in what software is deployed means that prediction has a big role to play in decision support. Leading the target is essential. There appear to be two alternatives here -- minimal change and maximal change.

On the maximal change side, Sandy Clark, et al., have pretty much shown [14] that software quality does not matter if you roll your code base often enough. The key is "often enough" meaning often enough that the reverse engineering opponents don't have time to get good at beating this version before it is replaced with another. It is a compelling idea, so let me quote Sandy on this:

Analysis of software vulnerability data, including up to a decade of data for several versions of the most popular operating systems, server applications and user applications (both open and closed source), shows that properties extrinsic to the software play a much greater role in the rate of vulnerability discovery than do intrinsic properties such as software quality. This leads to the observation that (at least in the first phase of a product's existence), software vulnerabilities have different properties than software defects.

We call the period after the release of a software product (or version) and before the discovery of the first vulnerability the 'Honeymoon', and show that familiarity with the system is the primary driver for the length of the honeymoon period. We also demonstrate that legacy code resulting from code re-use is also a major contributor to both the rate of vulnerability discovery and the numbers of vulnerabilities found; this has significant implications for software engineering principles and practice.

As a sort of corroboration, you might be interested to know that those High Frequency Trading applications change more or less constantly, they are all closed source, and they run entirely without firewalls or anything that anyone here would call a "security mechanism."

On the minimal change side, Andy Ozment and Stuart Schecter showed [15] that if you work long and hard on a code base, then the number of security vulnerabilities in that code base does decline over time. Quoting from their original paper:

We examine the code base of the OpenBSD operating system to determine whether its security is increasing over time. We

measure the rate at which new code has been introduced and the rate at which vulnerabilities have been reported over the last 7.5 years and fifteen versions.

We learn that 61% of the lines of code in today's OpenBSD are foundational: they were introduced prior to the release of the initial version we studied and have not been altered since. We also learn that 62% of reported vulnerabilities were present when the study began and can also be considered to be foundational.

We find strong statistical evidence of a decrease in the rate at which foundational vulnerabilities are being reported. However, this decrease is anything but brisk: foundational vulnerabilities have a median lifetime of at least 2.6 years.

Finally, we examined the density of vulnerabilities in the code that was altered/introduced in each version. The densities ranged from 0 to 0.033 vulnerabilities reported per thousand lines of code. These densities will increase as more vulnerabilities are reported.

So in these two extremes, minimal change and maximal change, we have measures that show, however indirectly, alternate paths to low exploitation opportunity for our opponents. In other words, the curve of exploitability has a peak in middle age while infancy is free of exploit and advancing maturity enjoys what might be called an acquired immunity. Similarly, it appears that for any given software package the version that is one rev off of current is the most attacked. That is an almost biologic observation -- you either want to stay in the center of the herd or disappear into the bush, that is keep up with revisions or run something that no one is (still) attacking.

For some software suppliers, the workfactor of constant release and constant remediation for a large userbase is better handled by the software as a service model. Putting aside the consumer-side lock-in, that is all well and good. Of course, people like you at meetings like this conference have no independent measures of the rate at which security flaws are introduced, found, or fixed in software as a service settings.

Auto-update of software might be said to be the most common software as a service offering. It is a great thing so long as it is capable of dealing with local anomalies. Generally speaking, you get a better result if you don't try to analyze too much, just replace the whole software package. However, if anyone else that doesn't like you ever gets control of your auto-update mechanism, then it will be hard to ever pick up the pieces well enough to truthfully say that you got back to where you were before the strike.

Just don't forget that total cycle time for a round of updates matters. The coming Smart Grid, is, after all, an application layered on top of the biggest machine in the world. Kelly Ziegler's numbers [16] indicate that should it be necessary to do a total update of the firmware for all U.S. households on a fully deployed Smart Grid it would take a year or so. What might we do differently?

• • •

A product is a security product when it has sentient opponents

Security involves making sure things work, not in the presence of random faults, but in the face of an intelligent and malicious adversary trying to ensure that things fail in the worst possible way at the worst possible time... again and again.

Perhaps we are imagining that the digital world is more dissimilar to the physical world than it really is. Perhaps the better way to express this is to say that when the opponent you face will simply try again and perhaps harder if his first approach is repulsed, then designing for failure is the core of your concern. When a street light drops a lens and misses you but having missed you the next street light doesn't cough up its whole light arm, there is no security issue. When you've whacked a dog with a stick and now it wants your arm even more, now you've got a security issue.

...

Earlier I spoke of trendlines as something that even a somewhat lossy measurement can still provide and, more to the point, that trendlines provide ordinal scales that are almost always sufficient for decision making. If you have a risk measure that is at least stable enough to yield a trendline, and you have a corresponding measure of some exposure, then you can look at what is called Relative Risk, namely the change in risk that a change in the exposure transmits. It is trivially calculated: Relative Risk is the probability of the risk given the presence of the exposure divided by the probability of the risk given the absence of the exposure. A Relative Risk greater than one simply means that the risk is more likely given the exposure.

I suspect some of you know all that already, and you may also know that dividing the Relative Risk minus 1 by the Relative Risk gives you the Attributable Risk Percent or ARP. The Attributable Risk Percent is important -- it is the portion of all risks that could have been avoided had the exposure been avoided. Attributable Risk Percent is my choice for a metric [18] with which to think about attack surfaces.

In their definitive paper, Manadhata & Wing [19] provided two important insights about relative measures of attack surface:

[I]f we create a newer version of a software system by only adding more resources to an older version, then assuming all resources are counted equally, the newer version has a larger attack surface and hence a larger number of potential attacks. Software developers should ideally strive towards reducing the attack surface of their software from one version to another, or if adding resources to the software (e.g., adding methods to an API), then do so knowingly that they are increasing the attack surface.

[I]f software developers increase a resource's damage potential and/or decrease the resource's [required] effort [to compromise] in their newer version, then all else being equal, the newer version's attack surface measurement becomes larger and the number of potential attacks on the software increases.

So, if you call feature expansion an "exposure," then measuring the Relative Risk of adding that feature delivers exactly what I've said that security metrics exist only to do: provide decision support -- in this case decision support for the question of whether the

feature is worth the inevitably increased Relative Risk. And if you have that Relative Risk, then you can ask the possibly more pertinent question of whether the Attributable Risk Percent is a tolerable price to pay for the feature.

Attack surface Relative Risk may well be old news with respect to the Windows and Linux operating systems, but Arik Hessendahl reported [20] four days ago that as of the second quarter of this year the majority of memory chips no longer go into PCs, they go into tablets, phones, embedded systems, and so forth. The same is true for software -- the number of new apps for Apple and Android combined is well over 1,000 per day. That is where the attack surface is growing and where the Attributable Risk Percent figure is most needed for decision support including, but not limited to, the Bring Your Own Device trend.

Turning to Manadhata's & Wing's section on "Lessons Learned,"

Choosing a suitable configuration, especially for complex enterprise-scale software, is a nontrivial and error-prone task. A system's attack surface measurement is dependent on the system's configuration. Hence assuming that vendors provide attack surface measurements for different configurations of their software, software consumers would choose a configuration that results in a smaller attack surface.

[I]n the maintenance phase, software developers can use the measurements as a guide while implementing vulnerability patches. A good patch should not only remove a system's vulnerability, but also should not increase the system's attack surface.

Both of these learned lessons might as well just have said that measurement of Relative Risk is necessary if rational risk management tradeoffs are to be made. But in the smartphone case, end-user configuration control and/or patch choice are impossible (and that impossibility is strictly intentional on the part of the network operator). Given that, would it be too much to ask the network operator to inform you what the Relative Risk between an upgraded system and the current system is? Of course it isn't (too much to ask), but does anyone care?

...

If you buy the argument that designing for failure is a central point in security design, then you will surely want to have the kind of instrumentation that when things do go badly you will be able to do forensic analysis that makes the event less likely to again occur. Speaking as a statistician: Get the data -- you can always throw it away later. Getting the data is what the Reference Monitor can do if you add logging to it. Getting the data is what those price-flickering HFT boxes are doing. Getting the data is what Network Flight Recorder did and what Net Witness is doing. And so forth. The less you can prevent failure the more you must not let it be silent, the more you must have forensic-readiness as part of your design. In John Tan's original formulation [21], forensic-readiness meant precisely two things:

Minimising the cost of forensics during an incident response

It is consistently measured, i.e., objective and repeatable

It has a units of measure, like dollars

It is relevant to decision making

To repeat, then, the most important thing the Software Assurance Community can do is to ensure that there is no silent failure. This means instrumentation, it means well designed surveillance regimes, it means an attention to the kind of metrics that come out of an airplane's flight data recorder, it means keeping things simple enough that there are fewer surprises, and it may mean changing how you think about how you make tradeoffs. Take Kernighan's warning to heart, and do not write code as cleverly as possible because if you do then you will not be smart enough to debug it. Because security is not composable (and may never be), be very careful where the code you reuse comes from, and reuse only what you need, the cheapness of the cloud notwithstanding.

=====

[2] Bernstein P: *Against The Gods: the Remarkable Story of Risk* Wiley, 1996; see also http://www.mckinseyquarterly.com/Strategy/Strategic_Thinking/Peter_L_Bernstein_on_risk 2211

[4] O'Dell M: personal communication

MEW-6AEaEdEcAAlpcNi ~E^ eei e-aAElmcEiO of Lt ECaEc-dOEEaEi ~KQMNVKri

- [6] Searching for a Speed Limit in High-Frequency Trading, New York Times, 8 Sept 2012
<http://www.nytimes.com/2012/09/09/business/high-frequency-trading-of-stocks-is-two-critics-target.html>
- [7] Hubbard D: *_How to Measure Anything_*, Wiley, 2010
- [8] Hoare CAR: "There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies and the other is to make it so complicated that there are no obvious deficiencies."
- [9] Kolsek M: personal communication
- [10] Davis D: personal communication
- [11] Anderson J: "Computer security technology planning study," Technical Report ESD-TR-73-51, AFSC, Hanscom AFB, Bedford, Mass., October 1972
- [12] The Orange Book, "Department of Defense Standard: Department of Defense Trusted Computer System Evaluation Criteria," DoD 5200.28-STD (Supersedes CSC-STD-001-83), 15 August 1983
- [13] Index of Cyber Security, <http://www.cybersecurityindex.org>
- [14] Clark S, Frei S, Blaze M, & Smith J: "Familiarity Breeds Contempt: The Honeymoon Effect and The Role of Legacy Code in Zero-Day Vulnerabilities," ACSAC, 9 December 2010
http://www.acsac.org/2010/openconf/modules/request.php?module=oc_program&action=view.php&a=&id=69&type=2
- [15] Ozment A & Schecter S: "Milk or Wine: Does Software Security Improve with Age?," USENIX Security Symposium, Vancouver, B.C., 31 July 2006 <http://research.microsoft.com/pubs/79177/milkorwine.pdf>
- [16] Ziegler K: "Smart Grid, Cyber Security, and the Future of Keeping the Lights On," USENIX Security Symposium, 13 August 2010
- [17] Anderson R: *_Security Engineering_*, Wiley, May 2001
<http://www.cl.cam.ac.uk/~rja14/bruce.html>
- [18] Geer DE: "Attack Surface Inflation," IEEE Security & Privacy, July/August 2011 <http://geer.tinho.net/ieee/ieee.sp.geer.1107a.pdf>
- [19] Manadhata PK & Wing JM: "An Attack Surface Metric," IEEE Transactions on Software Engineering, May/June 2011
<http://www.cs.cmu.edu/~pratyus/tse10.pdf>
- [20] Hessendahl A: "It's Official: The Era of the Personal Computer Is Over," Dow Jones, 19 September 2012
<http://allthingsd.com/20120915/its-official-the-era-of-the-personal-computer-is-over/>
- [21] Tan J: "Forensic Readiness," Secure Business Quarterly, July 2001 (dead link follows)

[22] Jaquith A: *_Security Metrics_*, Addison-Wesley, March 2007